TP1A: Detección de bordes

J. Enríquez - LU 36/08 - juanenriquez@gmail.com
N. Gleichgerrcht - 160/08 - nicog89@gmail.com
J. Luini 106/08 - jluini@gmail.com

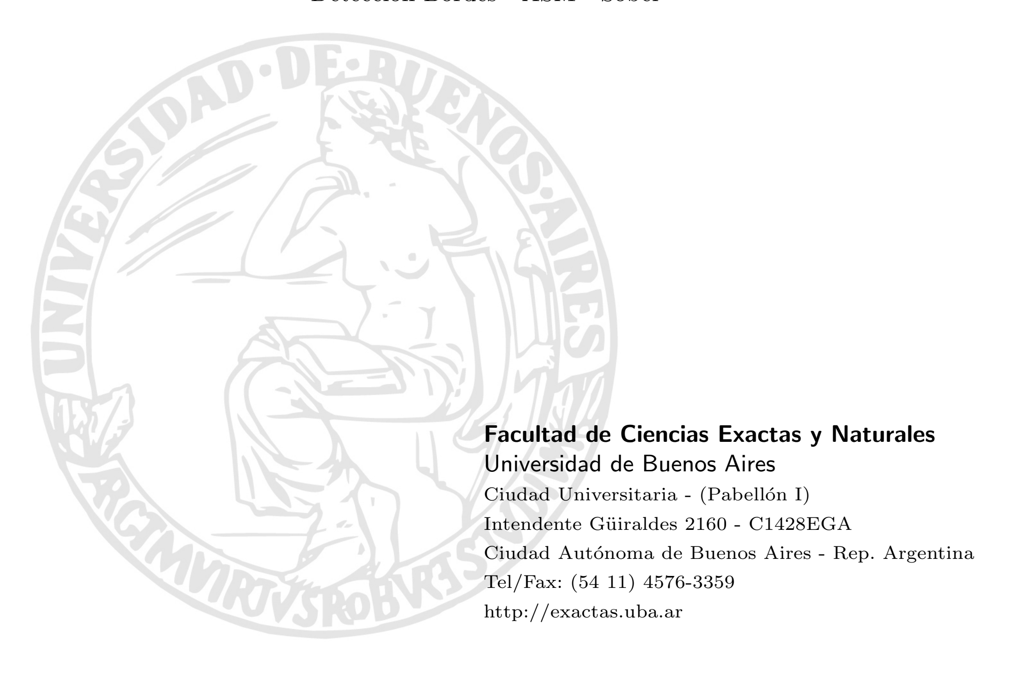
Resumen

En este trabajo nos proponemos implementar en lenguaje ensamblador, y usando la arquitectura básica de la IA-32, la detección de bordes en imágenes usando convolución. La técnica consiste en detectar las regiones de la imagen en las que la intensidad cambia abruptamente y graficar esos saltos en nuevas imágenes.

Describimos aquí las cuestiones surgidas durante la planificación del algoritmo y su implementación, tarea en la que el énfasis estuvo puesto en maximizar la eficiencia temporal. Además compararemos esa eficiencia con la de una implementación de la biblioteca de procesamiento de imágenes OpenCV y analizaremos las conclusiones surgidas de esa comparación.

Keywords

Detección Bordes - ASM - Sobel



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Introducción

La detección de bordes en una imagen consiste en hallar las zonas de la misma en donde el color cambia .ªbruptamente". Esta herramienta es utilizada tanto para compresión de archivos como para lograr efectos sobre las imágenes.

Una manera simple y muy útil de detectar bordes es calculando cómo varía la intensidad o luminosidad de una imagen entorno a cada uno de sus pixeles. Esto se logra tomando la imagen en escala de grises y aplicando en cada punto un operador de derivación, que es una matriz de números cuyo producto interno con el entorno del punto mide la variación de la intensidad en alguna dirección. Esta técnica se llama **convolución**.

Existen muchas matrices utilizadas para la convolución, con diferentes dimensiones y coeficientes. Las de mayores dimensiones proveen una detección más suave y menos precisa en la que se reducen los efectos del ruido".

En este trabajo hicimos un programa que permite aplicar distintos operadores de derivación a imágenes y visualizar los resultados en forma de nuevas imágenes en escala de grises. Las funciones de procesamiento fueron hechas en lenguaje ensamblador, utilizando la arquitectura "básica" de la `IA-32` con sus registros de propósito general. Cabe destacar que en el presente trabajo no se utilizaron algunas de las extensiones propias de la arquitectura mencionada como ser las instrucciones MMX, SSE, etc.

Medimos la performance de nuestras funciones, en cuanto a cantidad de clocks insumidos por el procesador, comparándolas entre sí y con una implementación de detección de bordes de la biblioteca de procesamiento de imágenes **OpenCV** (en particular la función `cvSobel`).

En este informe describiremos brevemente el programa realizado, discutiremos las cuestiones surgidas durante su desarrollo y expondremos los resultados y conclusiones extraídos.

2. Desarrollo

2.1. Programa

El programa que hicimos permite aplicar ciertos operadores de derivación a una imagen especificada por línea de comandos. Los operadores implementados son Roberts, Prewitt y Sobel. El primero tiene matrices de 2x2 y detecta bordes en diagonal. Los otros dos tienen matrices de 3x3 y detectan bordes verticales u horizontales.

Para cada operador solicitado el programa aplica la matriz correspondiente en X, luego en Y, y finalmente suma los resultados. Para el caso de Sobel, también permite aplicar convolución solamente en X o solamente en Y.

La imagen de entrada puede tener cualquier formato que la función `cvLoadImage` de *OpenCV* soporte; todos los formatos comunes están incluidos. Por cada operador utilizado se guarda una nueva imagen en escala de grises (con un sufijo en el nombre que indica el operador y la misma extensión y tamaño que la imagen de entrada) que grafica los bordes detectados.

Por último, el programa muestra por salida estándar la cantidad aproximada de clocks de procesador utilizados para la ejecución del algoritmo.

Las implementaciones están hechas en lenguaje ensamblador, aunque el programa también permite usar la implementación del operador Sobel de la biblioteca *OpenCV* (función `cvSobel`), así como también grabar la versión en escala de grises de la imagen de entrada. Por último, permite también aplicar una primera implementación del operador de Sobel que hicimos en C. Esta implementación fue realizada para ayudarnos a desarrollar el algoritmo.

El algoritmo consiste en aplicar la matriz correspondiente en X, saturar el valor a 0 y 255, luego aplicar la matriz en Y, saturar también su resultado, y finalmente sumar ambos valores y saturar la suma. Ese valor es el que se graba en la imagen generada.

También usamos *OpenCV* para cargar y guardar imágenes y para pasarlas a escala de grises.

2.2. Algoritmos

Si bien la detección de bordes requiere cálculos sencillos e intuitivos, hubo varios aspectos en los que surgieron dudas y posibilidades.

En un píxel en donde la intensidad de la imagen crece hacia la derecha, diremos que se trata de un borde "hacia la derecha". Si en ese punto la imagen en cambio oscurece hacia la derecha, diremos que es un borde "hacia la izquierda".

Notar que, en el caso de Prewitt o Sobel, la matriz de convolución en X (que detecta bordes verticales) arroja un resultado positivo en los bordes hacia la derecha y negativo en los bordes hacia la izquierda. Asimismo la matriz de convolución en Y (que detecta bordes horizontales) produce números positivos en bordes hacia abajo y negativos en bordes hacia arriba. Con las matrices de

Roberts pasa lo mismo, salvo que los bordes son positivos hacia el noroeste para la matriz X o hacia el noreste para la Y, y negativos hacia el sudeste o sudoeste respectivamente.

Hay muchas formas de tratar los valores númerico de la derivación en X y en Y de cada píxel para graficarlos. Con la implementación que hicimos en C de Sobel probamos varias de ellas. Llamemos dx y dy a los valores obtenidos aplicando derivación en X y en Y respectivamente.

Una opción es graficar un punto cuya intensidad refleje el módulo del vector (dx, dy). De esta manera se obtiene una imagen oscura con los puntos borde más claros. La imagen muestra los bordes en todas las direcciones, aunque no permite distinguir en qué sentido van. Es decir, un borde hacia la derecha se grafica igual que uno hacia la izquierda. Para el cálculo del módulo puede usarse la norma 2 o bien la norma 1, bastante más rápida de computar.

Otra transformación típica es saturar los valores a un mínimo y a un máximo, lo que permite visualizar los bordes a grandes rasgos sin importar todos los matices de variación. Por ejemplo se los puede saturar a 0 y a 255. Hay que notar que haciendo esto se desprecian todos los bordes negativos, por lo tanto sólo se grafican los bordes que van en un sentido. Si en cambio se suma 128 al valor antes de saturarlo, se puede generar una imagen en la que predomina el gris y donde los bordes positivos se muestran como zonas claras y los bordes negativos como zonas oscuras.

Las variantes que implementamos nosotros son las siguientes:

1. $\|(dx + dy)\|_1$
2. $dx + dy + 128$ saturado a 0 y a 255
3. saturación de dx y dy a 0 y 255, suma, y saturación de la suma a 255

La idea inicial que teníamos, sugerida en la consigna del trabajo, era implementar la primera variante, que grafica bordes en todas las direcciones. Finalmente, sin embargo, decidimos usar la tercera variante porque sus resultados coinciden con los arrojados por la implementación de Sobel de OpenCV, contra la cual deseábamos comparar performance. Esto quiere decir que las imágenes generadas por nuestro programa son oscuras, con zonas claras donde la imagen original tiene aumentos de intensidad hacia abajo o hacia la derecha (hacia el noroeste o hacia el noreste en el caso de Roberts). Los bordes que van en otras direcciones no se visualizan; así que es imposible saber dónde están o cuán abruptos son.

La variante 2 produce gráficos diferentes e interesantes de observar. Se pueden ver bordes en ambos sentidos, graficados de manera distinta. Los bordes positivos se muestran en blanco y los negativos en negro, sobre el fondo gris. Sin embargo no muestra bordes en todas las direcciones. De hecho, el efecto de sumar la derivada en X con la de Y es que los bordes se grafican en una única dirección (una diagonal). En los bordes perpendiculares a esta diagonal dx se anula con dy y por lo tanto estos bordes no se visualizan.

En el momento que optamos sumar las derivadas en X y en Y porque esa parecía ser la técnica usada por OpenCV, surgió una idea: en lugar de aplicar

la matriz de X, luego la de Y y sumar los resultados, sería más eficiente (y más sencillo de implementar) hacer una sola pasada usando la matriz suma de las dos matrices.

Sin embargo las imágenes generadas haciendo ese "truco"no se parecían todavía a las de OpenCV. Algunos bordes, pese a ser "positivos", quedaban ocultos. Lo que sucedía era que, usando la matriz suma, por ejemplo una derivada positiva en X se puede anular con una derivada negativa en Y. Esto hace que se vean casi exclusivamente los bordes que apuntan en dirección sudeste (o norte para Roberts).

Finalmente se concluyó que la técnica correcta (para generar dibujos como los de OpenCV) era saturar cada derivada parcial a 0 y a 255 y recién después sumarlas. Como la primera saturación elimina negativos, es imposible que las derivadas se cancelen, y se terminan visualizando todos bordes cuya dirección tiene alguna componente hacia abajo o hacia la derecha.

2.3. Implementación

Cuando hicimos la primera implementación (la del operador de Sobel hecha en C) nos topamos con un primer problema: la imagen resultante parecía mostrar los bordes, sin embargo estaba llena de puntos blancos, como si el algoritmo saturara ante el borde más suave. Finalmente resultó que el problema se solucionaba haciendo *casts* explícitos de los `char` obtenidos de las imágenes de OpenCV a `unsigned char`.

Dado que la correcta interpretación de las imágenes de OpenCV se realiza tratando los elementos del array `imageData` como `unsigned char`, todavía nos preguntamos por qué este campo del struct está declarado como `char*` (y no como `unsigned char*`).

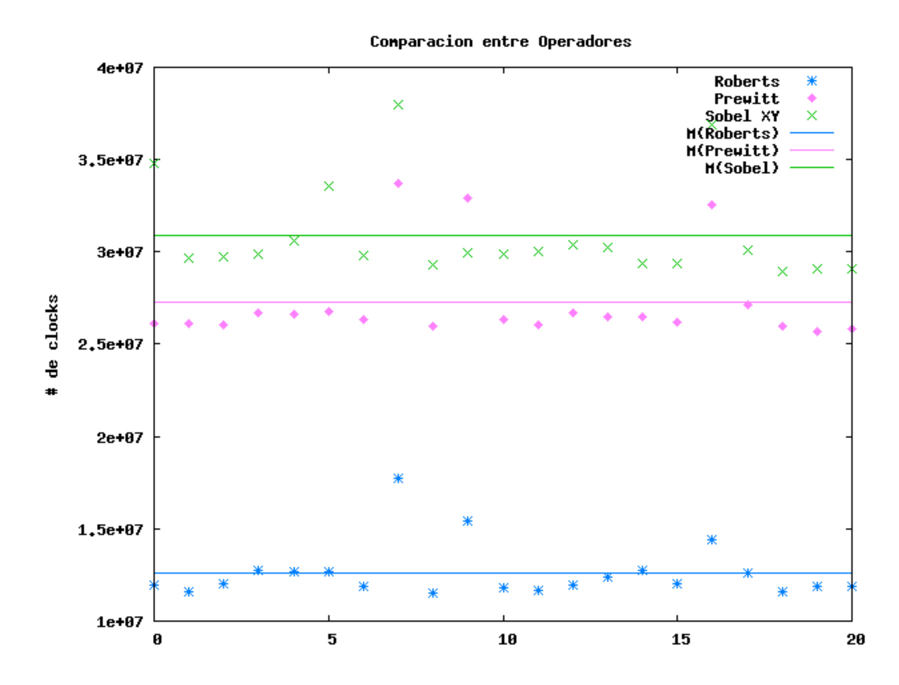
Una vez que tuvimos la primera versión correcta hecha en lenguaje ensamblador (que aplicaba derivación en X mediante el operador de Sobel) nos tuvimos que embarcar en la tarea de mejorar el código. Si bien funcionaba correctamente abusaba de los accesos a memoria ya que para cada píxel estudiado salvaba y recuperaba información usando la pila (instrucciones `push` y `pop`). Esto provino del "apuro" por terminar la implementación; sin embargo muy pronto logramos que el algoritmo utilizara sólo los registros, con lo que se lograron mejoras importantes de performance.

La versión final del trabajo todavía permite ejecutar el programa con esta implementación provisoria de Sobel. Como es posible verificar ejecutando el programa, la implementación que usa la pila es menos eficiente (toma aproximadamente el doble de tiempo que la otra).

3. Discusión

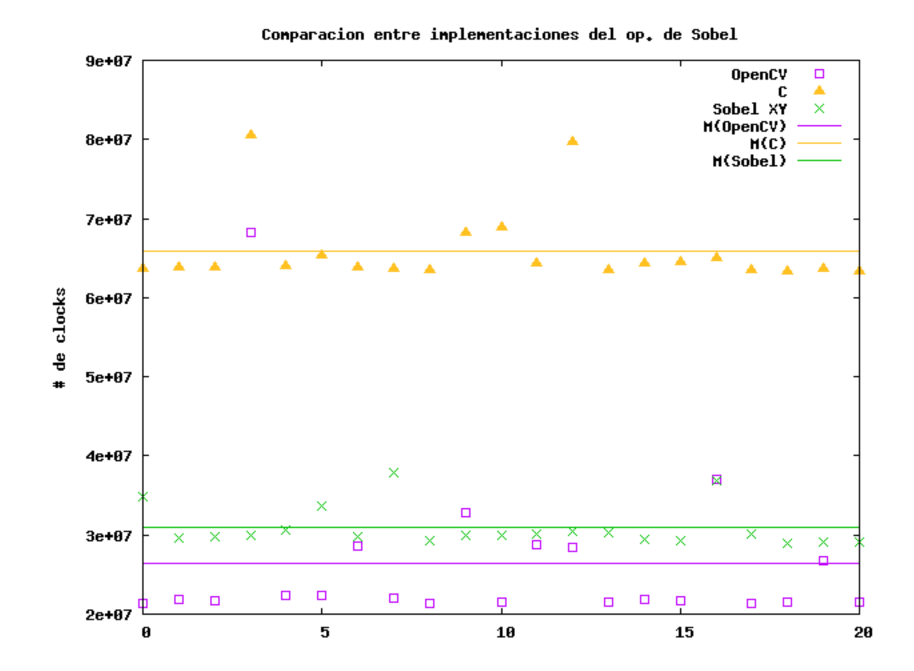
Tratándose de implementaciones en lenguaje en ensamblador, uno de las cuestiones que más tuvimos en cuenta y analizamos fue, por supuesto, la eficiencia de los algoritmos. Como ya dijimos, lo que hicimos fue medir la cantidad de clocks de procesador que cada función (incluso la de OpenCV y la que escribimos en C) consumen. Esto se logró mediante invocaciones desde C a la función `rdtsc` de lenguaje ensamblador.

El resultado que obtuvimos en ese sentido fue, como esperábamos, que nuestras implementaciones en ensamblador no alcanzaron la rapidez de la implementación de OpenCV, y que la implementación hecha en C fue aún más lenta. Los siguientes gráficos reflejan estas diferencias para imágenes de distintos tamaños.[1]



	Roberts	Prewitt	Sobel
Media	12618185	27276745	30886607
Desvío	1511854	2448652	2598392
Max	17743236	33694296	37954836
Min	11549460	25707576	28905552

[1] *Las ejecuciones fueron realizadas en una PC: MSI Wind U100 con procesador Intel Atom N270 - 1Gb de RAM y bajo el S.O. Ubuntu Netbook Remix 9.04*



	OpenCV	C	ASM
Media	26412756	65944312	30886607
Desvío	10563470	4921205	2598392
Max	68276028	80518500	37954836
Min	21301152	63370956	28905552

La diferencia de eficiencia entre la implementación en C y las hechas en ensamblador tiene un motivo claro y conocido. Por naturaleza, los lenguajes de programación de más alto nivel nos abstraen de muchas cuestiones de implementación, pero a la vez nos quitan control sobre esa implementación. Mientras en ensamblador usamos directamente los registros del procesador (memoria más rápida de la máquina) en C éstos se usan de manera interna y el programador trabaja con datos en memoria RAM.

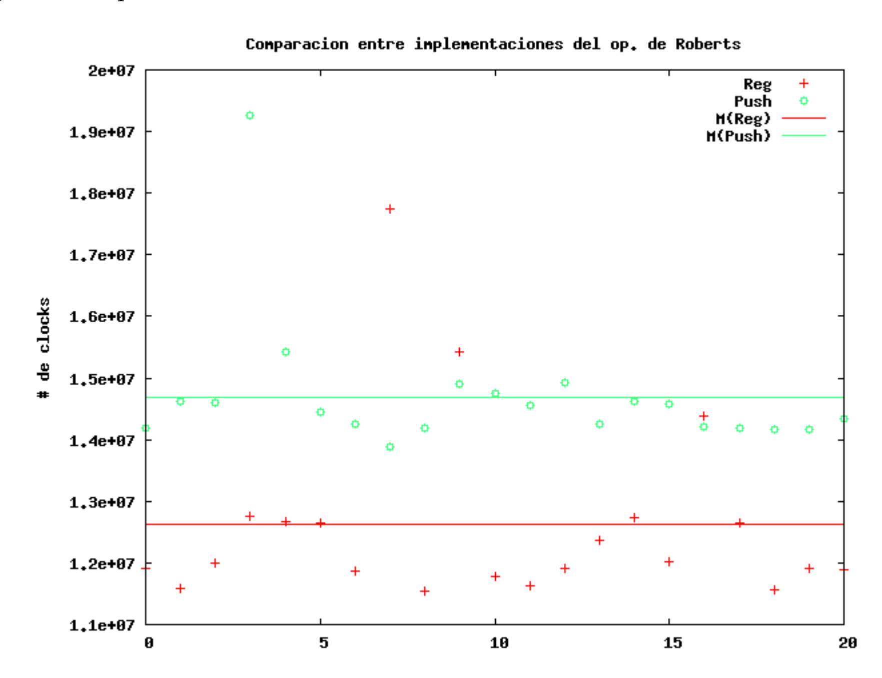
Esta razón bastaría para justificar las diferencias de rendimiento, pero no es la única: además, el sólo hecho de estar programando en un lenguaje de nivel superior hace que el programador ponga menos atención en los detalles de eficiencia y más en otros aspectos como la legibilidad del código o su reutilizabilidad.

En cambio, el motivo de la diferencia de rendimiento entre nuestras implementaciones en ensamblador y la de OpenCV resulta menos evidente. De hecho no estudiamos el código que utiliza OpenCV. Sabemos, sin embargo, que nuestras implementaciones no aprovechan al máximo las posibilidades de los procesadores, a diferencia de las de OpenCV que seguramente utilizan capacidades más avanzadas y/o específicas de los mismos como por ejemplo el uso de MMX, SSE, etc..

Un ejemplo que se nos ocurre es la saturación: nosotros, para cada píxel, verificamos si el resultado está por debajo de 0 o por encima de 255 para ajustarlo. Estos chequeos y ajustes consumen un tiempo significativo de ejecución; es claro que usando aritmética saturada nativa del procesador se mejoraría mucho la performance.

Otra característica que imaginamos se podría aprovechar es la paralelización: al tratarse de cálculos sencillos y repetitivos podrían venir bien las instrucciones de datos empaquetados.

También resulta interesante observar en qué medida se aceleró el algoritmo cuando eliminamos los accesos a memoria innecesarios que realizábamos al utilizar la pila en lugar de sólo los registros. El gráfico que sigue compara el rendimiento de la primera implementación en ensamblador que hicimos (que aplicaba Roberts en X usando la pila) con la definitiva (que minimiza los accesos a memoria y aplica Roberts en ambas direcciones). Es notorio que, aunque la primera implementación realiza menos cálculos pues aplica una sola matriz en lugar de dos, sea tanto menos eficiente por no aprovechar al máximo los registros del procesador.



	"Reg"	"Push"
Media	12618185	14692721
Desvío	1511854	1105330
Max	68276028	19269852
Min	11549460	13878600

4. Conclusiones

Como discutimos anteriormente, verificamos el hecho esperado de que nuestras implementaciones no alcanzarían la rapidez de las de OpenCV.

También es nuy notorio cómo se puede llegar a mejorar la performance de un algoritmo implementándolo a tan bajo nivel. Para imágenes de tamaños típicos la ejecución realmente parece instantánea.

Para mejorar aún más las implementaciones creemos que el camino es aprovechar características más avanzadas de los procesadores modernos, en particular -y como ya mencionamos- la paralelización de cálculos y la utilización de una aritmética saturada más cercana al nivel de hardware. Suponemos que siguiendo este camino podríamos acercarnos mucho a la eficiencia lograda por OpenCV, aunque no tenemos aún las herramientas para verificarlo.

5. Apéndice A: Manual de usuario

El programa realizado permite aplicar diferentes implementaciones de detección de bordes a imágenes. Tanto el nombre de las imagen fuente como los operadores se indican por línea de comandos de la siguiente manera (suponiendo que el ejecutable está en .exe/bordes”):

```
$ exe/bordes {fuente} {destino} {operadores}
```

Donde:

* `fuente`: es la imagen de origen; puede tener cualquier formato soportado por la función cvLoadImage de OpenCV; los formatos más comunes están soportados;

* `destino`: es el nombre de las imágenes de salida (generadas por el programa) SIN INCLUIR LA EXTENSIÓN; los archivos generados tendrán ese nombre más un sufijo que indica el operador utilizado, más la extensión; la extensión, el formato y el tamaño son obtenidos de la imagen de entrada;

* `operadores`: una o más claves separadas por espacio; cada clave representa una implementación particular de detección de bordes; las claves posibles y sus significados son los que siguen:

CLAVE	OPERADOR	DIRECCION	IMPLEMENTACIÓN	SUFIJO
r1	Roberts	XY	Ensamblador	_asm_roberts
r2	Prewitt	XY	""	_asm_prewitt
r3	Sobel	X	""	_asm_sobelX
r4	Sobel	Y	""	_asm_sobelY
r5	Sobel	XY	""	_asm_sobelXY
cv3	Sobel	X	OpenCV	_cv_sobelX
cv4	Sobel	Y	""	_cv_sobelY
cv5	Sobel	XY	""	_cv_sobelXY
c3	Sobel	X	C	_c_sobelX
c4	Sobel	Y	""	_c_sobelY
c5	Sobel	XY	""	_c_sobelXY
push	Roberts	X	Ensamblador usando pila	_asm_roberts(push)
byn	Escala de grises			_byn

Por ejemplo, si tenemos el ejecutable en `"exe/bordes"` y una imagen en `"pics/lena.bmp"`, llamando al programa como

```
$ exe/bordes pics/lena.bmp pics/lena byn r1 r2 r3 r4 r5 cv3 cv4 cv5 c3 c4 c5 push
```

se aplicarán todas las implementaciones disponibles y se generarán los archivos `"pics/lena_asm_roberts.bmp", "pics/lena_asm_prewitt.bmp"`, etc... Además el programa mostrará por salida estándar la cantidad aproximada de clocks de procesador insumida por cada implementación.